Train by Roman Kovbasyuk from The Noun Project

#005

Rails 102

Ticket to Intermediate Rails



Яосо

by xdite, Wayne

Table of Contents

Introduction

本書是根據 xdite 在 2011 的 邁向 Rails 高级新手 – 你所需要知道的一些知識 作為架構。其中部分章節標題略有不同，但仍可以與原文章相呼應，也歡迎協助校訂，謝謝。

本書目的：

以初學者能夠理解的語言深入介紹 Ruby on Rails 相關知識，讓初學者能夠更快上手。

本書將分為四大章節：

1. 關於 MVC
2. Rails 週邊知識
3. Ruby 基本常識
4. Ruby 寫作慣例

作者資訊

本書由 Rocodev 的 Yi-Ting Cheng (xdite) 與 Wayne Chu (wayne5540) 共同著作

原始碼

本書原始碼放置在 http://github.com/rocodev/rails-102/ ，歡迎直接 fork 修正內容錯誤。

Chapter1 - 關於MVC

MVC指是M(Model)、 V(View)、 C(Controller)，是軟體開發的一種架構模式，目的是將不同的行為切開來，對於程式碼撰寫與維護上可以比較有規則。而Rails的開發是遵循著MVC的架構的。

此系列將介紹在MVC中應該要知道的相關知識。

MVC三部曲：

Model

View

Controller

Model

Model是什麼？

Model是 繼承於 ActiveRecord 的 Class，所以每個 Model 都可以取用 ActiveRecord 裡面的 Method，例如我們有一個Model叫做 Post，這個Model會長這樣：

```
class Post < ActiveRecord::Base
end
```

而這個 Model 將會負責操作 `Posts` 的資料庫，由於Post Model繼承ActiveRecord，所以我們可以用ActiveRecord的method，例如：

```
# first就是ActiveRecord的Method
Post.first
# => 回傳第一個Post
```

什麼東西應該放在Model

只要是跟操作資料庫有關的行為基本上都應該儘量放在Model裡面實作，大部份情況是要操作已經叫出來的實例變數。（若對 Ruby 的變數不太了解請先參考Ruby系列教學）

has_many :through

`has_many :through` 主要是在建立多對多關聯資料庫的時候使用。

例如一個「球隊」可以有很多「球員」， 一個「球員」可以參加很多「球隊」， 這個就是多對多關係。

在這裏球員跟球隊都是一個 Model， 要建立多對多關係時我們會需要第三個 Model扮演連接的橋樑。

第三個 Model 就是「體育協會」， 負責登記每個球員所屬的球隊以及球隊目前的成員。如此一來我們就可以從體育協會得知每個球隊和球員的狀況（所以叫作「through」）。

所以「球員」throuth「體育協會」has_many「球隊」， 舉例：

models/player.rb

```
class Player < ActiveRecord::Base
  # 先告訴model我們在體育協會有很多筆資料
  has_many :sports_associations
 # 這些資料是要拿來判斷這個球員有參與多少球隊
  has_many :teams, :through => :sports_associations
end
```

models/team.rb

```
class Team < ActiveRecord::Base
  has_many :sports_associations
  has_many :players, :through => :sports_associations
end
```

models/sports_associations.rb

```
class SportsAssociations < ActiveRecord::Base
  #體育協會要對球員和球隊負責，所以體育球隊belongs_to球員和球隊
  belongs_to :team
  belongs_to :player
end
```

validation

validation是在 Model 內驗證資料的方法，用以確保資料符合我們的要求。

validation 運作的時機是在 SQL 指令執行之前，所以當 ActiveRecord 認為這筆資料未通過驗證，就不會將 Ruby 轉換為 SQL 語言寫入，而是傳回錯誤訊息。

validation常見的寫法有：

```
class Topic < ActiveRecord::Base
  # presence: true代表這個資料必須有值
  validates :title, presence: true
  # content最多只能200字
  validates :content, length: { maxmum: 200 }
end
```

會引發驗證機制的method有 `create` 、 `save` 、 `update` 等。

正常情況若validate fail時 create 會把object丟回來，update和save則是會回傳false，倘若想知道錯誤原因，可以在method後加上驚嘆號 `save!`

```
topic.save
# => false
topic.save!
# => ActiveRecord::RecordInvalid: Validation failed: Title can't be blank
```

參考資料

除了上述兩種驗證方式，尚有驗證文字格式、數字、唯一等等，可以查看 Rails Guide 得到更多資訊。

scope

scope 的作用就是將時常使用或是複雜的ORM語法組合成懶人包，這樣下次要用的時候只要把懶人包拿出來就可以了，舉例說明：

```
class Topic < ActiveRecord::Base
    scope :recent, -> { order("created_at DESC") }
end
```

上面這段code我們定義了`recent`這個scope，以後我們只要下`recent`這個指令就等於下`order("created_at DESC")`是一樣的。如此一來就可以讓程式碼更為簡潔。

使用情境

- 當有過於複雜的資料查詢
- 當有重覆使用的資料查詢

使用方式

沒帶參數的方式

```
class Post < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end
```

帶有參數的方式

```
class Post < ActiveRecord::Base
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

可以串接在一起，順序沒有影響

```
class Event < ActiveRecord::Base
  scope :published, -> { where(published: true) }
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

`Event.published.created_before(Time.now)`

參考資料

- http://guides.rubyonrails.org/active_record_querying.html

ids

collection_singular_ids是當我們做has_many時產生的method。

假設我們有 `Drink` 和 `Material` 兩個many-to-many的model 並希望使用者在新增Drink的時候可以同時選取Material做關聯

這時候的做法就是讓被選取的materials的id存成array，經由 `params[:material_ids]` 傳給drink controller rails就會根據 `material_ids` 幫我們建立關聯。

補充：

記得要在 `drink_controller.rb` 的strong params內加上 `material_ids: []` 使用simple_form時只要下 `f.association` 就可以了

```
<%=  simple_form_for @drink do |f| %>
  <%= f.association :materials, :as => :check_boxes %><br>
  <%= f.submit "Submit" %>
<% end %>
```

參考資料

- http://railscasts.com/episodes/17-habtm-checkboxes
- http://guides.rubyonrails.org/association_basics.html#has-many-association-reference

collect(&:id)

基本上跟map是一樣的東西，collect做的事情就是將array內的資料一一拿出來處理，處理完後再傳回array 所以`topics.collect(&:id)`就是把topics內所有的id都拿出來存成另一個array

ex:

```
topics = [{id:1, title:"topic1"}, {id:2, title:"topic2"}]
topics.collect(&:id)
# => [1,2]
```

稍微進階一點的做法：

假設我們要讓id都變成浮點數(雖然我想不到理由為什麼要變成浮點數，XDrz)

我們可以這樣下指令：

```
t.collect{|t| t.id.to_f }
# 或是更簡單一點：
t.collect(&:id).collect(&:to_f)
# 回傳都會是一樣的
# => [1.0,2.0]
```

參考資料

- http://stackoverflow.com/questions/5254732/difference-between-map-and-collect-in-ruby
- http://rubyinrails.com/2014/01/ruby-difference-between-collect-and-map/

includes

當我們操作關聯性資料時，可以使用includes將關聯資料先抓出來，這樣每次要調用資料時就不是去資料庫搜尋，而是從已經調閱出來的資料搜尋，可以提高效能。舉例說明：

假設「Board」has_many「Topics」，原本我們會這樣寫：

```
class Board < ActiveRecord::Base
  def
    @boards = Board.all
  end
end
```

這種情況下若我們在view裡面寫 `@board.topics`，假設我們共有10個board，系統會這樣幫我們找資料：

1.到資料庫裡找board_id = 1 的topic

2.到資料庫裡找board_id = 2 的topic

.....

10.到資料庫裡找board_id = 10 的topic

這樣的做法有個很弔詭的地方在於我們重複搜尋了資料庫10次，造成效能問題。

若加上 `includes` 後會是這樣：

```
class Board < ActiveRecord::Base
  def
    @boards = Board.includes(:topics).all
  end
end
```

這種時候我們若在view裡面寫 `@board.topics`，假設我們共有10個board，系統會這樣幫我們找資料：

1.到資料庫裡把topic全部抓出來

2.從抓出來的 topic中找到 board_id 相對應的topic。

這樣做我們就只需要搜尋一次，直接減少了資料庫的負擔。

counter_cache

`counter_cache` 是在做關聯性資料庫時計算資料量的一個方法。

照樣舉「Board」has_many「Topics」的例子來看 正常我們會寫類似下方的程式碼來算出這個 Board 內有多少個 Topics：

```
@board.topics.size
```

這代表每次我們要算數量的時候都得下一個sql指令去算有多少topics，一樣是會造成效能問題。

所以 Rails 內建了 `counter_cache` 的方法，我們只要在 Topic model 內加上 `counter_cache: true`

```
class Topic < ActiveRecord::Base
  belongs_to :board, counter_cache: true
end
```

然後在 Board 的資料庫內新增一個叫做 `topics_count` 的欄位

這樣以後當這個 Board內的 topics 有增減的時候，Rails就會幫我們增減 `topics_count` 的欄位，如此一來下次再下 `@board.topics.size` 的指令時 Rails就會預設去找 `topics_count` 的欄位，不用重新下 SQL指令計算。

自定 Counter Cache 欄位：

我們也可以覆蓋 Rails的預設規定，自己定義要增減的欄位名稱，例如把 `topics_count` 改成 `count_of_topics` ：

```
class Topic < ActiveRecord::Base
  belongs_to :board, counter_cache: :count_of_topics
end
```

STI

全名：單一表格繼承 STI(Single-table inheritance)

Rails Guide 對此的說明：

> a way to add inheritance to your models

簡單來講就是讓繼承的 submodel 可以擁有父類別的表格欄位且繼承父類別的方法。

在 Rails 慣例中只要加上`type`這個欄位在父類別的資料庫中就可以了

例：User有分 Native 跟 Foreigner

```
class User < ActiveRecord::Base
end
class Native < User
end
class Foreigner < User
end
```

這樣就可以了，可以新增 Native User

```
native = Native.create(:name => "foobar")
native.type # => "Native"
```

STI的使用時機是當我們需要一個擁有一樣特性但是不同行為的model時才使用。

參考資料

- http://thibaultdenizet.com/tutorial/single-table-inheritance-with-rails-4-part-1/

Polymorphic Assoiciaion

Polymorphic Associations是 Rails 內可以讓兩個不同的 Model 同時 has_many 一個 model的做法，最常見的情況就是要做留言系統的時候。

假設我們希望使用者可以針對文章留言，也可以針對看版留言，但是這些留言所需要的欄位都一樣，我們沒有必要新增像 `TopicComment` 和 `BoardComment` 這樣的model 只需要新增一個 `Comment` model 再用 Polymorphic Associations 關聯就可以了。

範例

Model的設定如下：

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end

class Board < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class Topic < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

而 Comment裡面 必須包含兩個欄位分別是 `commentable_id` 和 `commentable_type`

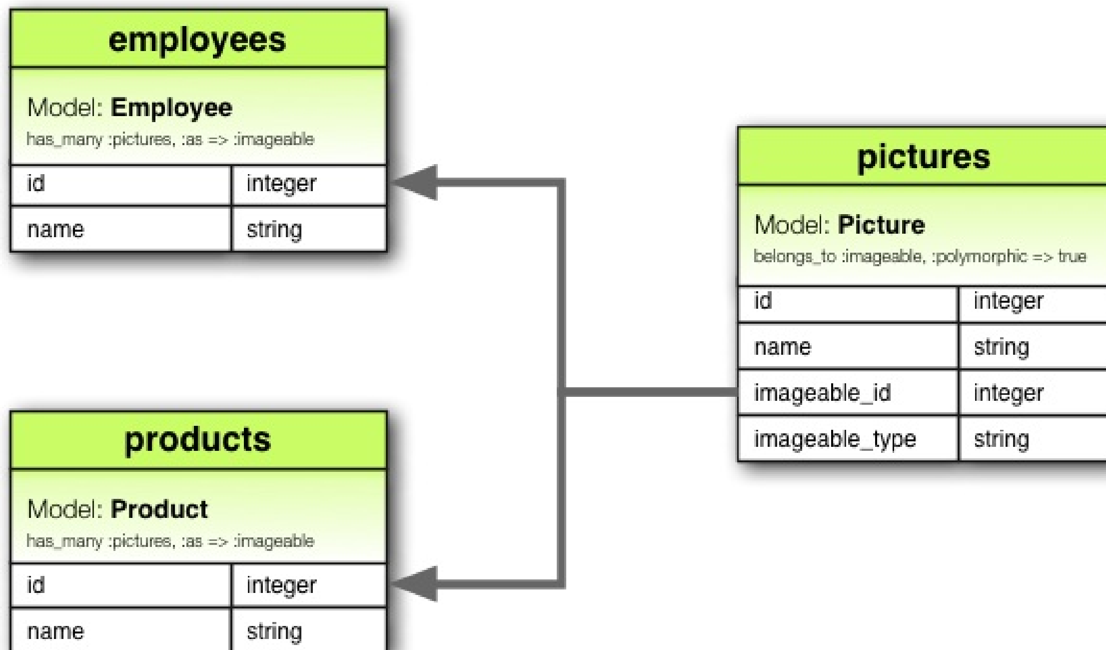
`commentable_type` 記錄 model 名稱

`commentable_id` 記錄該 model 下的資料id

Rails3之後的 migration 可以單獨寫 `t.references :imageable, polymorphic: true` 就會自動生成這兩個欄位。

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, polymorphic: true
      t.timestamps
    end
  end
end
```

作用示意圖




```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

參考資料

- http://guides.rubyonrails.org/association_basics.html#polymorphic-associations
- http://railscasts.com/episodes/154-polymorphic-association

View

什麼是View

View基本上就是負責處理視覺呈現的地方，在rails的開發中view就是負責放html碼的地方，也就是前端呈現。

什麼東西應該放在View

原則上View只應該出現跟前端呈現有關的東西，儘量不要把判斷以及邏輯相關的code放在View，例如以下的例子就是不好的做法：

```
<% if current_user %>
  <p><%= current_user.name %></p>
<% else %>
  <p>Not Logined</p>
<% end %>
```

這個例子中在view判斷使用者是否登入，且還對使用者讀取name的欄位 比較好的做法是將`if else`改寫在partial中，並把`current_user.name`用helper包起來。這樣html code的好讀性就可以大大提升。

請參考本章詳細介紹。

什麼是 helper

Helper 是一些使用在 Rails 的 View 當中，用 Ruby 產生/整理 HTML code 的一些小方法。通常被放在 `app/helpers` 下。預設的 Helper 名字是對應 Controller 的， 產生一個 Controller 時，通常會產生一個同名的 Helper。如 `PostsController` 與 `PostsHelper`。

使用 Helper 的情境多半是：

- 產生的 HTML code 需要與原始程式碼進行一些邏輯混合，但不希望 View 裡面搞得太髒。
- 需要與預設的 Rails 內建的一些方便 Helper 交叉使用。

使用 Helper 封裝程式碼可以帶給專案以下一些優點：

- Don't repeat yourself（DRY）程式碼不重複
- Good Encapsulation好的封裝性
- 提供 view 模板良好的組織
- 易於修改程式碼

```
module BoardsHelper

 # 回傳board的name，避免在view中做太多判斷
 def render_board_name(board)
     if board.present?
       board.name
   else
       "unknown"
   end
 end

# 常常重複的區段也可以寫進 helper，統一管理
 def render_board_name_path(board)
   link_to(board.name, board_path(board))
 end
end
```

在view中可以直接取用如下

```
<%= render_board_name_path(@board) %>
```

什麼是 partial

Partial 簡單說就是程式碼中的一小段，通常使用在 HTML 中讓 View的Code 可以更乾淨，將重複使用到的區塊切成獨立的 Partial，比方說頁首頁尾、表單、社群插件等等，讓任何一個頁面都可以讀取這段Partial而不用重複寫一次一模一樣的 Code。

使用情境

什麼時候應該將把程式碼搬到 Partial 呢？

*long template | 如果當檔 HTML 超過兩頁 *highly duplicated | HTML 內容高度重複 *independent blocks | 可獨立作為功能區塊

使用範例

假設我們常常需要在頁面產生 topics 的列表，就可以考慮將列表包裝成 partial，這樣每個頁面需要時只要 render 這個 partial 就可以了。

`_topic_list.html.erb`

```
<ul>
<% @topics.each do |topic| %>
  <li># <%= topic.id %></li>
  <li>Topic Name: <%= link_to(topic.title, topic_path(topic)) %></li>
  <li>Description: <%= topic.content %></li>
<% end %>
</ul>
```

這樣頁面就會變得很簡單：

`index.html.erb`

```
...
<%= render "topic_list" %>
...
```

什麼是 collection partial

正常的 Partial 只會將內容 render出來一次，collection partial 則會自動 count 我們給予的物件，並render count的次數，可以省去在 partial 內再寫 block 或 helper的繁瑣，讓 partial 更簡潔。

延續上一題的例子，我們可以改寫成：

`_topic_list.html.erb`

```
<li># <%= topic.id %></li>
<li>Topic Name: <%= link_to(topic.title, topic_path(topic)) %></li>
<li>Description: <%= topic.content %></li>
```

然後這樣render：

`index.html.erb`

```
...
<ul><%= render :partial => "topic_list", :collection => @topics, :as => :topic %></ul>
...
```

如此一來就不用在view裡面包block了。

什麼東西應該放在 partial / 什麼東西應該放在 helper

partial 負責處理大片的 HTML code，或是之後要利用 ajax render 出來的片段。

helper 則負責處理跟邏輯判斷有關的東西。

`topics_helper.rb`

```
module TopicsHelper
# 需要邏輯判斷的工作留給helper
def render_published_topic_name_path(topic)
  link_to(topic.name, topic_path(topic))  if topic.is_published?
end
```

`_topic_info.html.erb`

```
# partial只需要負責html的部分
<table>
  <thead>
    <tr>
      <td>Topic name</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td><%= render_published_topic_name_path(topic) %></td>
    </tr>
  </tbody>
<table>
```

yield in view

yield 就是會被替換成樣板的地方，基本上所有的 `html.erb` 檔案最後就是顯示在 `<%= yield %>` 的地方，這樣做的好處是可以將網站的版型固定，只在需要出現內容的地方用 yield 引進來就可以了。

通常 Rails專案中都會有一個 `application.html.erb` 的檔案，我們的 `html.erb` 檔就是被引入這裡

`application.html.erb`

```
<!DOCTYPE html>
<html>
<head>
  ...
</head>
<body>
  <div class="container">
    # erb檔案就是被引入這裡
    <%= yield %>
  </div>
  #可以指定要yield哪個區塊
  <%= yield(:javascripts) %>
</body>
</html>
```

稍微進階一點的做法：

另外我們也可以指定要yield哪個區塊，我們可以用 `content_for` 的方式讓content替換掉 `<%= yield(:javascripts) %>` ：

```
<%= content_for :javascripts do %>
    #content here
<% end %>
```

不應該放在 view 裡的東西

邏輯判斷

```
<% if topic.present? %>
  <%= @topic.title %>
<% else %>
  Unknown
<% end %>
```

ruby block

```
<% @topics.each do |topic| %>
  #content
<% end %>
```

撈 Model資料的code

```
<%= Topic.find(params[:id]).title %>
```

form

Form 表單是給使用者輸入資料的地方，預設是使用 `POST` 方法送出資料，再由 Rails 決定是新增或是修改。

Rails內最常見的表單 helper 就是 `form_for`， 可以幫我們針對特定model以及model內的欄位送出資料。

```
<%= form_for @topic do |f| %>
  <%= f.label :name %>
  <%= f.text_field :name %>
  <%= f.submit %>
<% end %>
```

其中如 `label`、 `text_field` 和 `submit` 也是 form 的 helper，幫助產生對應的 HTML tag。

Nested form

Nested form 可以將有關聯的 model attribute 放在同一個 form 裡面一起建立或更新。

例如這次的實作是一個商品有很多種規格，我希望在新增商品的時候就可以同時新增完規格（ product has_many specs ），Controller相關的設定解說可以參考 Strong Parameters 章節的相關解說。

Model如下：

```
class Product < ActiveRecord::Base
  has_many :specs, :dependent => :destroy
  accepts_nested_attributes_for :specs
```

```
class Spec < ActiveRecord::Base
  belongs_to :product
end
```

如此一來後端的設定就算是OK了，接下來是前端form的部分，在此用simple_form作為例子，rails guide內也有原生form的教學可以 參考此連結

如果是用 simple_form 就是使用 `simple_field_for` 的這個helper即可

view/admin/products/_form.html.erb

```
<%= simple_form_for [:admin, @product] do |f|  %>
  <%= f.input :name, :required => true %>
  <%= f.input :description, :required => true %>
  <%= f.input :price %>

  <!-- nested form從這裡開始 -->
  <%= f.simple_fields_for :specs do |spec| %>
    <%= spec.input :name %> <!-- spec的name欄位 -->
    <%= spec.input :detail %> <!-- spec的detail欄位 -->
  <% end %>
  <!-- nested form結束 -->

  <%= f.button :submit, "Submit", :disable_with => "Submiting..." %>
<% end %>
```

這樣就完成了最簡單的nested form了。

常見 Helper

Rails 最令其他 Ruby Web Framework 羨慕的就是內建的很多方便 Helper。舉幾個省下很多重造輪子功夫的 Helper ：

simple_format

可以處理使用者的內容中 \r\n 自動轉 br 和 p 的工作

auto_link

可以處理使用者的內容中若有連結就自動 link 的工作。（ Rails 4 已移除內建）

truncate

使用者輸入的內容若過長可以指定多少字後就自動砍掉並加入 “....”

html_escape

使用者輸入的內容若有 html tag為了怕使用者輸入惡意 tag 進行 hack。會進行自動過濾。

(Rails 3 之前要手動加 h 過濾，在 3.0.0+ 後預設 escap，不想被 escape 可以用 `raw` unescape)

這些東西若自己寫 parser 處理不知道要花費多少精力還不一定濾的徹底。卻都是 Rails 預設內建 Helper。

基礎建設 Helper

Rails 內建的某些 Helper 是為了與其他更棒的基礎建設整合：

stylesheet_link_tag 與 image_tag

有些開發者覺得，這東西還要用 Helper 嗎？直接貼 HTML 不是也一樣會動嗎? 有什麼差別。差別可大了。

```
<%= stylesheet_link_tag "abc","def",:cache => true %>
```

這一個 Helper可以在 production 環境時自動幫你將兩支 CSS 自動壓縮成 一支 applicatiion.css。直接實現了 Yahoo Best Practices for Speeding Up Your Web Site中minify HTTP reqesut 的建議。而在 Rails 3.1 之後甚至還會自動幫你 trim 與 gzip。

完全不需要去在 deploy process 中 hook 另外的 compressor 就可以達到。

image_tag

至於 image_tag 有什麼特別的地方?

```
<%= image_tag("xxx.jpg") %>
```

Rails 可以幫忙在 asset 自動在後面上 query string。若網站有整合 CDN 架構時，以自動處理 invalid cache 的問題。而 Rails 也有選項可以實作 asset parallel download 的機制，一旦打開，站上 的 asset 也會配合設定，亂數吐不同來源的 asset host 實做平行下載。

輕鬆就可以把網站 Scale 上去。

form_for

form_for 也是 Rails 相當為人稱讚的一個利器。Rails 的表單欄位是綁 model(db 欄位)。（解決資料庫映射問題)

開發方便(`Post.new(params[:post]` 直接收參數做 mapping)之外，也內建了防 CSRF (protect_from_forgery)的防禦措施。

Helper Patterns

Helper 是以 module 的形態存在的，在 Rails內 負責寫 View 要用到的method。

所以設計時能夠保持「簡單」且「可重複使用」是最重要的。

通常我們要使用 Helper 的時機是 View 有太多髒亂的code的時候，所以有時候會想把所有的髒 code 都寫進同一個 helper method裡面，這是不對的。

比較好的做法是應該把這些髒 code 拆成很多不同的小部分，再針對不同的部分判斷是要放在 partial 還是 helper。

我們也可以在 partial 內搭配 helper，這也是讓 view 變乾淨的好方法。

由於要讓 helper 保持「簡單」且「可重副使用」，在設計 helper 時應該讓h elper只「做一件事情」。（例如:判斷是否能編輯文章、回傳文章的連結按鈕）

這樣做的好處是這個 helper就會變得重複使用性很高（畢竟很多地方都需要判斷是否能編輯文章），就不會變成為了某個特定的 view 寫了一個 helper 卻只能用到一次，落實 Don't Repeat Yourself 的 Rails精神。

以下一些常見的 helper 設計方法舉例：

1. 儘量避免太多html code（無論ruby的或純html）

2. View裡面有太多讓人難以理解的判斷時，裝在helper裡面並語意命名，讓view更能理解

```
<% if current_user && current_user = @topic.user_id %>
  #show shomething
<% end %>
```

可以改成

```
<% if editable?(current_user) %>
  #show shomething
<% end %>
```

推薦參考文章：

- http://blog.xdite.net/posts/2011/12/09/how-to-design-helpers
- http://blog.xdite.net/posts/2011/12/10/how-to-design-helpers-2
- http://blog.xdite.net/posts/2012/01/16/how-to-design-helper-3

helper_method 與 view_context

helper_method

在 controller 裡面的 method 不能在 view 裡面用。

也就是在

```
class ProductsController
  def find_cart
    @cart = Cart.find(session[:cart_id])
  end
end
```

View 裡面不能用

```
<%= find_cart.items %>
```

拉這個 cart 出來直接用。

如果你要在 controller 和 view 都能拉現在的購物車，必須要用 helper_method 宣告這是一個 controller 級的 helper。

```
class ApplicationController
  helper_method :current_cart
  def current_cart
    cart = Cart.find(session[:cart_id])
    return cart
  end
end
```

這樣你就能在 View 裡面用 current_cart。

```
<%= current_cart.items %>
```

或者是 Controller 裡面也能用 current_cart。

```
class ProductsController
  def add_to_cart
    current_cart.items << @product
  end
end
```

view_context

在 helper 裡面的 method 不能在 controller 裡面用。 也就是在

```
class ProductsController
  def show
    @page_description = truncate(@product.desc, :lenght => 50 )
  end
end
```

是不會動的。

如果要在 controller 裡面取用系統內建的 Rails View Helper，或自定義的 View Helper。 必須要用 view_context 去調用。

```
class ProductsController
  def show
    @page_description =  view_context.truncate(@product.desc, :lenght => 50 )
  end
end
```

小結

但基本上還是建議在 View Helper 與 Controller 的 code 不要互相混來呼叫來呼叫去。讓 View 歸 View，Controller 歸 Controller。若真有業務上的需求需要「到處都可以用」。建議寫 Module 掛在 lib 用 mixin 技巧混入。

Controller

什麼是 Controller

Controller 主要是扮演橋樑的角色，負責跟 Model 要資料並把資料傳給 View。另外一方面也會接收 View 傳來的各種 HTTP request傳給 Model。

什麼東西應該放在Controller

基本上屬於「過程中應該被處理」的動作都應該放在 Controller，比方說有一個 View 需要前20筆的 products資料，我們不可能把所有的 products 都丟給 View ，然後再在 View 裡面判斷前20筆，這樣會很悲劇。所以在這種情況下，這個 View 對應的 Controller 就要負責「跟Model拿資料」以及「只拿20筆資料」的動作。

這個感覺有點像是球團、球員和經紀人三者的關係：

- 球團扮演的角色是 Model，負責出錢、辦比賽。
- 球員扮演的角色是 View，負責拿錢、比賽。
- 而經紀人就是 Controller，負責幫球員跟各個球團議價，讓球團跟球員能夠專注於自己的事情。

所以只要抱著「Model 處理資料庫，View處理前端呈現」這樣的原則，就知道什麼東西應該放在controller了。

Filters : before_action, after_action, around_action

`before_action` 的意思就是要求 Rails在run controller下的 action 前要先跑指定的method。相對的`after_action` 就是跑完 action 後才要跑的method，至於`around_action` 就是之前之後都要跑(嘖嘖，真貪心)。

以下為`before_filter` 的示範：

```
class TopicsController < ApplicationController

  before_action :find_board

  def index
    @topics = @board.topics
  end

  def find_board
    @board = Board.find(params[:id])
  end
end
```

我們在 TopicsController 上方要求它在每次執行 action 前都要`find_board`， 所以就不用在 index 裡面再定義一次`@board`， 如此一來若有很多action就不會有重複的程式碼產生。

此外我們也可以限定filter要作用的action，例如：

```
before_action :find_board, :only => [:index]
```

這樣 Controller在執行時就只會針對 index 這個 action 執行 `find_board`。

Strong Parameters

Rails 的 Form 綁 Model 設計非常直觀直覺，表單欄位直接對應到 Model 欄位。但也因此，容易被 Hack 猜中關鍵欄位的慣例，在瀏覽器自造欄位入侵測試，因而產生安全疑慮。

在 Rails 4 時，維護團隊發明一套方法能夠有效地解決這個問題。這套機制就是 Strong Parameters。

Strong Parameters 的想法是若 params 沒有被 permit，就是非法欄位，無法被直接送進 create / update method 更新。

使用方法

```
class PeopleController < ActionController::Base
  def update
    person.update_attributes!(person_params)
    redirect_to :back
  end

  private
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

Nested Attributes with Strong Parameters

在有些情況我們會需要做 Nested Form，這時候就需要將 Nested Form 的 Attributes 也加到 Strong Parameters 內。

假設每一個 Person Model 有都可以有一頂 Hat，在新增 Person 的時候用 Nested Form 同時新增 Hat 的顏色。我們應該這樣做：

1.在 Person Modle 中宣告我們可以接受 hat 的 attributes：

```
class Person < ActiveRecord::Base
  has_one :hat
  accepts_nested_attributes_for :hat
end
```

```
=begin
  Hat Database Information:
    id:integer
    color:string
    created_at:datetime
    updated_at:datetime
=end
class Hat < ActiveRecord::Base
    belongs_to :person
end
```

2.在 Controller 中指定接收的 Attributes

由於我們在 People Controller 內的 Strong Parameters 並不包含 color 這個 attributes，所以在新增 Hat 的時候就無法把 color 寫進資料庫，這時候需要加上 `hats_attributes: []` 這個 Attributes，並且將要傳的 Hat Attributes 寫進 Array 中：

```
class PeopleController < ActionController::Base
  def update
    person.update_attributes!(person_params)
    redirect_to :back
  end

  private
    def person_params
      params.require(:person).permit(:name, :age, hats_attributes: [:color])
    end
end
```

這樣就可以通過 People Controller 的驗證了。

render :template

render 的中文意思是「渲染」， 所以 render template就是「渲染」指定的模板，render的特性是不跑 controller action， 直接將該 action下 預設的模板傳出來， 我們也可以自己指定要哪個特定的模板。

舉例說明：

render 特定的模板

render views/foo/bar.html.erb

```
render "foo/bar"
```

render 同一個 controller 下的 action 的 view，用 symbol和string都是一樣的：

```
render :foobar
render "foobar"
```

render :layout

Rails 預設的 layout 是 `app/view/layouts/application.html.erb` 這個檔案。

但有時候我們會希望預設的版型不一樣，比方說我們的 admin頁面 head 內不希望加上GA和一些有的沒的追蹤script。

這時候我們就可以建立一個新的layout版型 `app/view/layouts/admin.html.erb`

只要在controller中指定使用admin layout即可：

```
class AdminsController < ApplicationController
  layout "admin"
end
```

稍微進階一點的做法：

我們也可以指定某個 action 要使用admin layout：

```
class AdminsController < ApplicationController
   layout "admin", :only => :new
   # 另外也可以在render的時候就指定要使用哪一個layout
   def show
      render :layout => "admin"
   end

   # 甚至可以指定模板再指定layout
   def index
    render :template => "others/weired_topics", layout: "admin"
  end
end
```

render :text

render 純文字。若是要把複雜的code包在js裡面的時候，可能會出現單雙引號打架之類的問題，所以用 render 的方式吐出 string 也許是一種方法。

```
render "嗨，自己的文字自己render"
```

render options

Rails 接受以下四種 render options：

:content_type

Rails預設會找 `text/html` 類型的檔案（除非我們有下 `:json` 或 `:xml` 的選項）。

但我們也可以藉由 `:content_type` 設定其他的檔案形態，例如：

```
render file: filename, content_type: "application/rss"
```

註記：content_type 類型可 Google `MIME搜尋`

Rails官方對content_type的說明

> By default, Rails will serve the results of a rendering operation with the MIME content-type of text/html (or application/json if you use the :json option, or application/xml for the :xml option.). There are times when you might like to change this, and you can do so by setting the :content_type option

:layout

文章上方已解釋過，可參考render :layout的部分

:location

location header是HTTP通訊協定回應時幫client重新定位的方法，當client丟一個request到server，我們可以丟回location這個header將client導到別的地方，舉例：

```
render xml: photo, location: photo_url(photo)
```

這段code的意思是告訴 client 端說 photo 的 xml檔要從 `photo_url(photo)` 這邊拿，所以假設我們有個 ajax script要來拿這個檔案，它就會跑到 `photo_url(photo)` 拿。

:status

指定server要丟什麼樣的http request status給client

```
# 以下兩種寫法都可以
render status: 500
render status: :forbidden
```

參考資源

- Rails Guides 內有一份status列表

redirec_to 與 render

`redirec_to`通常使用在要讓使用者跳轉頁面的時候，會執行指定頁面的controller action。

而`render`則是將指定頁面的樣板拿出來而已，並沒有執行controller。

通常`render`的使用時機是讓使用者回到同一個頁面，例如表單填寫不完全時再重回表單填寫頁，這樣做的原因是render會傳模板給使用者，而這個模板在使用者第一次送出表單時就已經被存起來了，所以render同一個模板的時候就會保留剛剛使用者打的表單資料，不用全部重打。

相反的若使用`redirect_to`跳轉到同一個表單頁面就會是一個全新的模板，不會有任何送出前填寫的資料，所以適合在跳轉到不同頁面時使用。

respond_to 與 respond_with

respond_to可以用來回應不同的資料格式，如果使用者要get www.example.com/ex.xml，rails會先尋找`ex.xml.erb`和`ex.xml.buulder`，最後會找`ex.html.erb`的檔案。

```
def index
  @topics = Topic.all
  respond_to do |format|
    format.html
    format.xml { render :xml => @topics }
    format.json { render :json => @topics }
  end
end
```

respond_with則是rails3之後才有的做法，可以先告知controller respond_to支援哪幾種檔案形態，如此一來在action內只要寫respond_with就可以了，因此可將上方的例子寫成下面的樣子：

```
respond_to :html, :xml, :json
def index
  @topics = Topic.all
  respond_with(@topics)
end
```

respond_with也可以寫成block，並在block內重新定義預設的render行為：

```
respond_to :html, :xml, :json
def index
  @topics = Topic.all
  respond_with(@topics) do |format|
    format.html { redirect_to root_path }
  end
end
```

builders

builders是一種 Template Handler，跟 erb 是很像的東西，erb 會幫我們把內容轉化為瀏覽器看得懂的 HTML 或 js 等，而 builder也是做一樣的事情，只是大家習慣使用erb作為 HTML 和js的handler，使用 builder作為 XML、RSS、Atom 的 handler。

補充：

> Builder templates are a more programmatic alternative to ERB. They are especially useful for generating XML content. An XmlMarkup object named xml is automatically made available to templates with a .builder extension.

所謂的 more programmatic alternative to ERB 應該是指它是以更程式的邏輯去 generate 出 XML file，所以更能展現 tag 間彼此的相對關係的意思，見下方例子：

`app/views/topics/show.xml.builder`

```
xml.topic do |t|
  t.title @topic.title
  t.content @topic.content
end
```

會產出這樣的xml

```
<topic>
  <title>Topic Title</title>
  <content>Topic Content Here</content>
</topic>
```

JSON builder

可以參考 jbuilder 生成 JSON 格式的資料。

RSS Builder

RSS、Atom 跟 XML 是很像的東西，因為 RSS 、 ATOM 都是基於 XML 格式的延伸應用，所以 Build 的方式跟 XML 一樣，唯一需要注意的就是一些 RSS 規範中需要有的 Tag ， 例如要有 rss version 的 tag ， 並把要呈現的內容放在 channel tag 中：

app/views/topics/index.rss.builder

```
xml.instruct! :xml, :version => "1.0"
xml.rss :version => "2.0" do
  xml.channel do
    xml.title "Rails102"
    xml.description "Intermediate to Rails"
    xml.link root_url
    for topic in @topics
      xml.item do
        xml.title topic.title
        xml.description topic.content
        xml.pubDate topic.created_at.to_s(:rfc822)
        xml.link board_topic_url(topic.board_id, topic)
        xml.guid board_topic_url(topic.board_id, topic)
      end
    end
  end
end
```

補充：**Generate RSS URI** 的方法

首先要讓 controller 可以 respond_to rss， 並且將 layout 預設為 false

```
class TopicsController < ApplicationController
  def index
    @topcis = Topic.all
    respond_to do |format|
      format.html
      format.rss { render :layout => false }
    end
  end
end
```

之後再 html view 內加上 RSS 連結就完成了。

```
<%= link_to "+RSS", posts_url(format: "rss") %>
```

Asset Pipeline

Asset Pipeline 是一套讓開發者很方便能夠削減 (minify) 以及 壓縮 (compress) Javascript / CSS 檔案的框架。它同時也提供你直接利用其他語言如 CoffeeScript / SASS / ERB ， 直接撰寫這些 assets 的可能性。

(assets 指的是 stylesheets / javascripts / images 這些靜態檔案。)

提升前端速度的優勢

一般在做前端提速時，我們通常會依據 Yahoo 提供的 Best Practices for Speeding Up Your Web Site 這篇文章提及的一些技巧對網站優化，諸如：

- Minimize HTTP Requests
- Use a CDN
- Split Components Across Domains
- Gzip Components
- Minify JavaScript and CSS
- CSS Sprites

在很久之前的版本，能夠使用 Rails 內建機制達成的只有這三項：

Minimize HTTP Requests

```
<%= stylesheets_include_tag "aaa", "bbb", :cache => true %>
<%= javascrupt_link_tag "ccc", "ddd", :cache => "customized-functions" %>
```

將數支檔案打包成一支，降低 HTTP requests。

Use a CDN + Split Components Across Domains

`config/enviroments/production.rb`

```
config.asset_host = "cdn%d.example.org"
```

使網站的靜態檔案從以下網址

- http://cdn0.example.org
- http://cdn1.example.org
- http://cdn2.example.org
- http://cdn3.example.org

平均動態提供，達到網頁高速載入的效果。

滿足實戰需求

但雖然已內建這些機制，但是對於一般有較大吞吐量的網站，還是不敷使用。

不只打包，更需壓縮

現在的網站設計師，多半會使用一些現成的 CSS 框架，如 Bootstrap 來做快速 prototyping。而網站逐漸擴充功能，一個網站的靜態檔案數量與所佔容量也會隨著水漲船高。有時候光 CSS 與 JavaScript 的檔案，加起來就足足有 1MB 以上的 size。

Rails 原先內建的功能，充其量只有將這些檔案「打包」，並沒有將檔案「壓縮」的作用。

若要談加速下載，唯有進行「壓縮」(trim / uglify / gzip) 才能達到真正減肥提升速度的目標。

字面上的「壓縮」其實還分幾種作法:

- 把不要的空白 / 不必要的注釋 去掉 (trim)
- 把檔案的變數以單個字母取代 (uglify)
- 將檔案壓成 gz

滿足主流 CDN invalid cache 需求

舊有的 Rails asset 是採用 query string 的方式，來達到 invalid browser cache 的作用。如：

```
<%= image_tag("example.gif") %>
```

會生成

```
<img src="/example.gif?1234567" />
```

query string 數字的演算法，是根據該 assets 更新的時間去定義。這樣每次 deploy 網站時，可以保證： assets 只要被更動過，query string 都會不一樣，而 browser 就將之視為不同的 URI，重新下載。

不過相當不幸運的，不是每一家 CDN 都支援 query string。

再者，stylesheets 內的 images，並不在 query string 的守備範圍內，也就是如果開發者使用了

```
background: url("logo-bg.gif");
```

當替換了 CSS 內的背景圖原始圖檔，或者是 CDN 不支援 query string。通常每 deploy 一次，你就必須上 CDN panel 去手動 invalid assets，並沒有辦法讓使用者的 browser 自動清除。

Fast by Default

這些都是相當難處理的問題。因此 Rails 在版本 3.1 後開發了 Asset Pipeline 架構來自動解決這些實際會發生的問題：

- 壓縮提速
- Fingerprinting Assets

並且倡導使用先進的 Asset 語言和框架，如 SCSS / Compass / CoffeeScript 結合 Rails 做更有效率的開發。

Sass / SCSS

Asset Pipeline 提供了內建直接使用 Sass 撰寫 CSS 的功能。你也許會生出這樣的疑惑：什麼是 Sass？ Why should I care?

Sass (Syntactically Awesome Stylesheets) 原先是內建在 Haml 中的一個副功能。

Haml

要談 Sass，就不得不先來談 Haml 這個 project。 Haml 全名為 HTML Abstract Markup Language，它是提供網頁設計師撰寫 HTML 的另外一條途徑。

透過 Haml，你可以很快的使用它的 syntax 寫出結構穩固的 HTML。

網頁設計師經常有一個煩惱：在撰寫 HTML 時要是忘記關一個 TAG，在瀏覽器中整個版面可能就會大爆炸，而這樣的 Bug 卻是很難被抓出來的。

Haml 主要就是讓開發者，能夠使用縮排的方式撰寫 HTML，輕鬆做到永不忘記關 Tag 的效果。以下是 Haml 範例：

```
%h1= "Hello World"
```

產生出來的 HTML 就會長這樣

```
<h1>Hello World</h1>
```

而

```
%ul{:id => "list", :class => "menu"}
  %li= "Item 1"
  %li= "Item 2"
  %li= "Item 3"
```

會產生出來這樣的 HTML

```
<ul id="list" class="menu">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
 </ul>
```

使用 **Haml** 撰寫 **HTML** 的好處

Haml 是需要使用縮排撰寫，再行 compile 的 markup language。它可以讓你：

阻絕撰寫出錯誤結構的 **HTML TAG** 的機會

只要 syntax 一錯誤，編譯就無法成功。利用這樣的特性，很容易阻絕寫出會在瀏覽器因為 TAG 結構錯誤而難以 debug 出的 HTML。

輕鬆調整排版

在網頁設計開發階段，若要大幅調整 HTML 結構，對網頁設計師也是很傷腦筋的一件事。因為大幅的搬動 HTML，通常有時候會造成：「少剪到一個 TAG」或 「改了開頭 TAG，卻忘了改關閉 TAG 」的憾事。

在 Haml 中，這些狀況都不會發生。因為 Haml 本身就屬於「塊狀結構」、「自我 close」。因此不論怎樣搬動和調整，只要符合縮排，幾乎都不會爆炸。

使用 Haml 撰寫 HTML 的壞處

如此 powerful 的 markup language 為何沒有風行？反倒是原先屬於副功能的 Sass 大紅特紅。原因就在於 Haml 的特性：不只需要被機器 compile，它也需要被人腦 compile。

HTML 本身就是一門相當直觀的 markup language。在撰寫 Haml 時，排版雖然相當輕鬆。但接手維護 Haml 版面的人，卻通常痛苦不堪。因為「非常不直觀」。

這也是 Haml 的反對者，批評最力的地方。

多數人無法接受維護不直觀的「任何東西」，加上撰寫 Haml 需要另外學習特殊的 syntax。沒有壓倒性的好處，一般人是不會貿然進行技術投資的。這也是為什麼 Haml 始終處是小眾技術的主要原因。

Sass / SCSS

拉回來談 Sass，Sass 原先是附屬在 Haml 裡面的一個副功能。這也是 sass-convert 這個指令必須要安裝 haml 這個 gem 才能使用的原因。

Sass 的原理，是讓開發者可以透過「縮排」的方式去撰寫維護 CSS，同樣可以避免忘記關 TAG 而大爆炸的糗事。

而因為 CSS 的結構特性，造成了 Sass 與 Haml 截然不同的命運。多數人反對 Haml，是因為 Haml 反而造成了 HTML 在閱讀上的不直觀。

而 Sass 的語法

```
.content
  margin: 2em 0
  h1
    font-size: 2em
```

產生出

```
.content{
 margin: 2em 0;
}
.content h1{
  font-size: 2em;
}
```

反倒讓 CSS 的維護變得直觀了。接觸 Sass / SCSS 後的不少開發者甚至認為，縮排 block 的撰寫方式才是 CSS 在被發明時應該被設計出來的樣子。

現在撰寫 CSS 的方式，有一個絕大缺點：只要在結構上涉及巢狀或多個 class，維護者就必須複製貼上 style。不少人認為這真是愚蠢至極且煩人透頂的設計。

其他便利功能

Sass 也提供了其他便利功能，如變數、函數、數學、繼承、mixin …等等功能。

在進行網頁 protyping 時，更改全站配色或者是直接提供兩個以上的設計，對設計師來說是家常便飯的事。

但更改全站配色卻是相當麻煩的一件事，因為「尋找 + 全數取代」，並不能保證最後會有正確的結果。很有可能：你更改了所有 CSS 中涉及連結的顏色，卻發現在全數取代的過程中，不小心也改到邊框的顏色。

若能使用變數去指定特定 style 的顏色，該有多好。

變數 (Variables)

```
$border-color: #3bbfce
$link-color: #3bbfcf
.content
  border-color: $border-color
  a
    color: $link-color
```

生成

```
.content{ border-color: #3bbfce; }
.content a{color: #3bbfcf; }
```

數學

在調整區塊寬度時，你也希望：每次調整寬度時，可不可以不要每次都按計算機，再全數手動修改...

```
.content
  width: (500px/2);
生成

``` css
.content{ width: 250px; }
```

內建函式

在調整顏色亮度時，你希望可否無需再開調色盤，直接改 CSS 就讓 style 暗一點？

```
$color = darken(#800, 20%)
.content
  background-color: $color
```

生成

```
.content{ background-color: #200; }
```

這都還只是 Sass 所提供的當中一小部分基礎功能而已，卻足以讓網頁設計師驚艷十足了。加上撰寫維護時十分直觀，這也難怪為何 Sass 只是 Haml 中的副功能，後繼的聲勢浪頭卻遠高於 Haml 本身。

SCSS

那 SCSS 又與 Sass 有什麼差別，他們看起來好像是類似的東西？

是這樣的，Sass 原先使用的縮排，對於網頁設計師來說，還是相當不直觀。而且實務上也不能直接將舊有的 CSS 直接貼進 Sass 中。其實還是存在一定的不方便度。也因此 Sass 進行了進化，改良了 syntax，而 Sass 3 後來就被稱為 SCSS (Sassy CSS)。

它的 syntax 與 CSS 原有的 syntax 完全 compatible，使用了 { } 去取代原先的縮排方式。

比如說原有的
```

```
.content
  margin: 2em 0
  h1
    font-size: 2em
```

在 SCSS 中變成了

```
.content{
  margin: 2em 0;
  h1 {font-size: 2em }
}
```

在撰寫上，更加無比的直觀，同時也能將舊有的 CSS 直接貼進去，完全沒問題！SCSS 更新增了不少關於 CSS3 的 feature 函式。

就拿我最愛的背景漸變色來說好了，原先要做漸變色，CSS 必須要這樣寫：

```
#linear-gradient {
  background-image: -webkit-gradient(linear, 0% 0%, 100% 100%, color-stop(0%, #ffffff), color-stop(100%, #dddddd));
  background-image: -webkit-linear-gradient(left top, #ffffff, #dddddd);
  background-image: -moz-linear-gradient(left top, #ffffff, #dddddd);
  background-image: -o-linear-gradient(left top, #ffffff, #dddddd);
  background-image: -ms-linear-gradient(left top, #ffffff, #dddddd);
  background-image: linear-gradient(left top, #ffffff, #dddddd);
}
```

因為你必須支援所有的 Browser。

但在 SCSS 中，一行就搞定了！

```
#linear-gradient { @include background-image(linear-gradient(left top, white, #dddddd)); }
```

Compass

Sass : extend / mixin / import

在 Sass 一章，我們只提到了一些基礎的部分。更強大的其實是這些功能

- extend
- mixin
- include

extend

extend 可以讓你將某些已經寫好的樣式，直接套入另外一組樣式中重複使用。

```
.warning{
  border: 1px;
  color: red;
}
.serious-warning{
   border: 1px;
   color: red;
   font-weight: bold;
}
```

而使用 extend，只需這樣寫，就可生成上述 CSS 了：

```
.warning{
  border: 1px;
  color: red;
}
.serious-warning{
   @extend .warning;
   font-weight: bold;
}
```

mixin

可以預先寫好自定要被 mixin 的 function，需要用時 include 進來

```
@mixin rounded-top {
  $side: top;
  $radius: 10px;
  border-#{$side}-radius: $radius;
  -moz-border-radius-#{$side}: $radius;
  -webkit-border-#{$side}-radius: $radius;
}
```

要用時 include 進來

```
#navbar li { @include rounded-top; }
#footer { @include rounded-top; }
```

import

可以把太長的 SCSS 拆開成一個一個的 partial，要用時 import 進來。

`style.scss`

```
//需要被 import 的檔案必須以 _開頭命名，如 _rounded.scss
@import "rounded";
```

有了 extend / mixin / import，加上 variable，加上支援部分 Ruby syntax。

聰明的你，應該想到這些元素集合起來可以作什麼？

沒錯，就是造 CSS Framework！

Compass = Sass framework powered by community

Sass 提供了一堆基本武器可以讓許多開發者造出自己的最佳實踐，再讓其他人可以重複利用。而 Compass 就是這些實踐產物的集合體。

Compass 自己的 core 充滿了 killer features，比如說：

CSS3 helpers

http://compass-style.org/reference/compass/css3/

撰寫漸層色與圓角框不再是令人頭大的難事。

漸層色 Background Gradients

http://compass-style.org/examples/compass/css3/gradient/

圓角框 Border radius:

http://compass-style.org/examples/compass/css3/border_radius/

CSS Pie

CSS3 Pie : http://compass-style.org/reference/compass/css3/pie/

不費吹灰之力實作 CSS Sprites

記得我們在 Asset Pipeline 中提到的 CSS Sprites 技巧嗎？它原是 Minimize HTTP Requests 中的一招，實作方式是將 CSS 中的小圖合併成較大的一張，再透過 CSS 定位的方式呼叫。比如說這個 CSS 中原先需要用到十張小圖，合併到只剩一張，就變得非常節省 HTTP requests（因為光是發出 requests 也相當耗時）。

問題是，負責 Scaling 的 server guy 很喜歡這招，但網頁設計師卻對這件事相當感冒。因為太麻煩了。試想，url('xxx.gif') 到處貼多方便呀，誰喜歡合併成一張，然後仔細精算定位？後續維護又怎麼辦呢？

Well，Compass 幫你「自動解決了問題」！http://compass-style.org/help/tutorials/spriting/

你可以把原先所有要用的小圖，通通丟進同一個資料夾。讓 Compass 幫你自動合併並算出位置！

```
@import "icon/*.png";
@include all-icon-sprites;
```

Compass 會幫你自動合併並算出位置：

```
.icon-sprite,
.icon-delete,
.icon-edit,
.icon-new,
.icon-save   { background: url('/images/icon-s34fe0604ab.png') no-repeat; }
.icon-delete { background-position: 0 0; }
.icon-edit   { background-position: 0 -32px; }
.icon-new    { background-position: 0 -64px; }
.icon-save   { background-position: 0 -96px; }
```

毛骨悚然的強大。

CoffeeScript

CoffeeScript 本身也是一種程式語言，開發者可以透過撰寫 CoffeeScript，編譯產生 JavaScript。它的語法有點像是 Ruby 與 Python 的混合體。

不得不坦承，當初 Rails 3.1 選擇同時納入 SCSS 以及 CoffeeScript 作為預設支援時，開發者是一半歡欣一半困惑的。歡欣的是納入 SCSS，困惑的是引入 CoffeeScript。

SCSS 讓撰寫 CSS 變得無比的直觀，並藉由 Compass 引入了不少 killer features。但使用 CoffeeScript 卻沒有這麼多壓倒的好處，有些人也形容，撰寫 CoffeeScript 需要腦袋內建 compiler …

好像用 Python 在寫 Ruby，但實際上寫的是 JavaScript

這是 CoffeeScript 的一段範例

```
name = "Rails";
greeting_words = "Hi, #{name}"
say_hi = (name) ->
  if name
    "Hello, #{name}"
  else
    greeting_words
```

編譯之後會變成

```
var greeting_words, name, say_hi;
name = "Rails";
greeting_words = "Hi, " + name;
say_hi = function(name) {
  if (name) {
    return "Hello, " + name;
  } else {
    return greeting_words;
  }
};
```

greeting_words 用的是 Ruby 語法

```
greeting_words = "Hi, #{name}"
```

say_hi 內的 if / else 則使用 Python 縮排

```
  if name
    "Hello, #{name}"
  else
    greeting_words
```

這是 CoffeeScript 的另外一段範例

```
jQuery ->
  $('#content').focus
```

編譯之後會變成

```
jQuery(function() {
  return $('#content').focus;
});
```

目的：The beautiful way to write JavaScript

乍看之下，CoffeeScript 給人一種詭異感覺：它讓寫 JavaScript 這件事變得更複雜？但也更簡單？...似乎有點矛盾。

Plurk 前創辦人 amix 曾寫過一篇這樣的 post：「CoffeeScript: The beautiful way to write Javascript」來這樣形容 CoffeeScript：「以更漂亮的方式撰寫 JavaScript」。

他認為目前 JavaScript 存在幾種問題：

- JavaScript 是 functional language
- 雖然是 OOP，但卻是 prototype-based 的
- JavaScript 是 dynamic language，更像 Lisp 而不是 C/Java，但卻用了 C/Java 的語法。
- 名字裡面有 Java，但卻和 Java 沒什麼關係。
- 明明是 functional & dynamic laungaue，更偏向 Ruby / Python，卻使用了 C / Java 的 syntax，原本可以是一門很美的語言，卻活生生的變成了悲劇。

而 CoffeeScript 的誕生，原因就是就是為了扭正這樣的局面，重新讓寫 JavaScript 這件事也可以變得「很美」。

CoffeeScript 做了幾項努力：

改善 Syntax

JavaScript 目前使用的是與本身型態不搭嘎的 C / Java 語法。CoffeeScript 改採用了深受 Lisp 影響的 Ruby + Python 的混合語法。

借鑑其他語言的好處

CoffeeScript 從其他語言借了不少好處，寫起來更方便。

Array (From Python)

```
# Array
list = [1, 2, 3, 4, 5]
# list comprehensions
cubes = (math.cube num for num in list)
# array slicing
copy = list[0...list.length]
# array generators
countdown = (num for num in [10..1])
```

String (From Ruby)

```
author = "Wittgenstein"
quote  = "A picture is a fact. -- #{ author }"
```

生成

```
var author, quote;
author = "Wittgenstein";
quote = "A picture is a fact. -- " + author;
```

多行 String 也沒問題

```
mobyDick = "Call me Ishmael. Some years ago -
never mind how long precisely -- having little
or no money in my purse, and nothing particular..."
```

這些例子若是直接使用 JavaScript 實作，肯定痛苦個半死…

Good Parts

CoffeeScript 留下了 JavaScript 的 Good Parts，而在設計上極力消除 JavaScript 原生特性會產生的缺點，例如：

消除到處污染的全域變數

開發者在寫 JavaScript 時，常不自覺的使用全域變數，導致很多污染問題。而透過 CoffeeScript 生出來的 JavaScript，變數一律為區域變數（以 var 開頭）

Protected code

使用 CoffeeScript 撰寫 function，產生出來的 JavaScript 必以一個 anonymous function: function(){}(); 自我包裹，獨立運作不干擾到其他 function。

使用 **->** 和 **indent**(縮排) 讓撰寫 **function** 更不容易出錯

在撰寫 JavaScript 時，最令人不爽的莫過於 function(){}();，這些複雜的括號和分號稍微一漏，程式就不知道死在哪裡了….

CoffeeScript 自動產生出來的 JavaScript 能夠確保括號們絕對不會被漏掉。

更容易偵測 **syntax error** 並攔阻

JavaScript 在 syntax error 時，非常難以偵測錯誤，幾乎是每個程式設計師的夢靨。而 CoffeeScript 是一門需要 compile 的語言，可以藉由這樣的特性擋掉 syntax error 的機會。

實作物件導向變簡單了

Javascript 是一種物件導向語言，裡面所有東西幾乎都是物件。但 JavaScript 又不是一種真正的物件導向語言，因為它的語法裡面沒有 class（類別）。

在 JavaScript 中，我們要實作 OOP 有很多種方式，你可以使用 prototype、function 或者是 Object。但無論是哪一種途徑，其實都「不簡單」。

但 CoffeeScript 讓這件事變簡單了，比如以官網的這個例子：

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5

class Horse extends Animal
  move: ->
    alert "Galloping..."
    super 45

sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"

sam.move()
tom.move()
```

容易建構 class，也很 Ruby Way

```
class Animal
  constructor: (@name) ->
```

自動生成

```
var Animal;
Animal = (function() {
  function Animal(name) {
    this.name = name;
  }
  return Animal;
})();
```

易於在 class 內增添function

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."
```

自動生成

```
var Animal;
Animal = (function() {
  function Animal(name) {
    this.name = name;
  }
  Animal.prototype.move = function(meters) {
    return alert(this.name + (" moved " + meters + "m."));
  };
  return Animal;
})();
```

實作繼承

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5
```

自動生成

```
var Animal, Snake;
var __hasProp = Object.prototype.hasOwnProperty, __extends = function(child, parent) {
  for (var key in parent) { if (__hasProp.call(parent, key)) child[key] = parent[key]; }
  function ctor() { this.constructor = child; }
  ctor.prototype = parent.prototype;
  child.prototype = new ctor;
  child.__super__ = parent.prototype;
  return child;
};
Animal = (function() {
  function Animal(name) {
    this.name = name;
  }
  Animal.prototype.move = function(meters) {
    return alert(this.name + (" moved " + meters + "m."));
  };
  return Animal;
})();
Snake = (function() {
  __extends(Snake, Animal);
  function Snake() {
    Snake.__super__.constructor.apply(this, arguments);
  }
  Snake.prototype.move = function() {
    alert("Slithering...");
    return Snake.__super__.move.call(this, 5);
  };
  return Snake;
})();
```

更容易撰寫測試

因為實作物件導向非常容易，所以撰寫測試這件事也更容易了。所以不少 Node.js 的套件或者是 JavaScript 的 Library 後來都改以 CoffeeScript 去實作。

Chapter 2 - 關於Rails

Ruby on Rails

Ruby on Rails，簡稱Rails，是一個使用Ruby語言寫的開源Web應用框架，它是嚴格按照MVC結構開發的。它努力使自身保持簡單，來使實際的應用開發時的代碼更少，使用最少的配置。

Rails的設計原則

1. 「不做重複的事」（Don't Repeat Yourself）
2. 「慣例優於設定」（Convention Over Configuration）

資料來源： http://zh.wikipedia.org/wiki/Ruby_on_Rails

RESTful on Rails

RESTful

Representational State Transfer，簡稱 REST。是 Roy Fielding 博士在2000年他的博士論文中提出來的一種軟體架構風格。目前是一種相當風行的 Web Services 實現手法，因為 REST 風格的 Web Services 遠比傳統的 SOAP 與 XML-RPC 來的簡潔。在近年，幾乎所有各大主流網站的 API 都已採用此種風格進行設計。

What is REST?

REST 提出了一些設計概念和準則：

1. 網路上的所有事物都被將被抽象成資源 (resource)
2. 每個資源對應一個唯一的 resource identifier
3. 通過通用的介面 (generic connector interface) 對資源進行操作
4. 對資源的各種操作不會改變 resource idetifier
5. 所有的操作都是無態 (stateless) 的

對照到 web services 上來說：

- resource indentifier 是 URI
- generic connector interface 是 HTTP

RESTful Web Services

RESTful Web Services 是使用 HTTP 並遵循 REST 設計原則的 Web Services。它從以下三個方面資源進行定義：

- URI，比如：http://example.com/resources/。
- Web Services accept 與 return 的 media type，如：JSON，XML， YAML 等。
- Web Services 在該 resources 所支援的一系列 request methid : 如：POST，GET，PUT 或 DELETE。

以下列出在實現 RESTful Web 服務時HTTP請求方法的典型用途。

GET

資源	GET
一組資源的URI，比如http://example.com/resources/	使用給定的一組資源替換當前整組資源。
單個資源的URI，比如http://example.com/resources/142	獲取 指定的資源的詳細信息，格式可以自選一個合適的 media type（如：XML、JSON等）

PUT

資源	PUT
一組資源的URI，比如http://example.com/resources/	列出 URI，以及該資源組中每個資源的詳細信息（後者可選）。
單個資源的URI，比如http://example.com/resources/142	替換/創建 指定的資源。並將其追加到相應的資源組中。

POST

資源	POST
一組資源的URI，比	在本組資源中創建/追加一個新的資源。 該操作往往返回新資源的

如http://example.com/resources/	URL。
單個資源的URI，比如http://example.com/resources/142	把指定的資源當做一個資源組，並在其下創建/追加一個新的元素，使其隸屬於當前資源。

DELETE

資源	DELETE
一組資源的URI，比如http://example.com/resources/	刪除 整組資源。
單個資源的URI，比如http://example.com/resources/142	刪除 指定的元素。

制約即解放

DHH 在 Discovering a world of Resources on Rails 提到一個核心概念：Constraints are liberating 。

很多剛踏入 Rails 這個生態圈的開發者，對於 REST 總有股強烈的反抗心態，認為 REST 是一個討厭的限制。但事實上，DHH 卻認為引入 REST 的制約卻反為 Rais 開發帶來了更大的解放，而這也是他引入這個設計的初衷。

維護性：解決了程式碼上的風格不一

在傳統的開發方式中，對於有效組織網頁應用中的程式碼，大家並沒有什麼共識。所以很可能在專案中會出現這種風格不一致的程式碼：

```
def add_friend
end

def remove_friend
end

def create_post
end

def delete_post
end
```

透過 REST 的包裝（對於 resource 的操作），可以變得更簡潔直觀，且具有統一的寫法。

```
class FriendShipController
  def create
  end

  def destroy
  end
end

class PostController
 def create
 end

 def destroy
 end
end
```

介面統一：**Action as Resource**

大眾最常對 REST 產生的一個誤解，就是以為 resource 只有指的是 data。其實 resource 指的是：data + representation (表現形式，如 html, xml, json 等)。

在網頁開發中，網站所需要的表現格式，不只有 HTML 而已，有時候也需要提供 json 或者 xml。Rails 透過 responder 實現了

在 Resource Oriented Architecture (ROA，一組 REST 架構實現 guideline) 中有一條：「A resource can use file extension in the URI, instead of Content-Type negotiation」。

Rails 透過 responder 的設計，讓一個 action 可以以 file extension (.json, .xml, .csv ...etc.) 的形式，提供不同類型的 representation。

```
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.xml
  def index
    @posts = Post.all

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @posts }
    end
  end
```

開發速度：透過 CRUD 七個 action + HTTP 四個 verb + 表單 Helper 與 URL Helper 達到高速開發

REST 之所以能簡化開發，是因為其所引入的架構約束。Rails 中的 REST implementation 將 controller 的 method 限制在七個：

- index
- show
- new
- edit
- create
- update
- destroy

實際上就是整個 CRUD。而在實務上，web services 的需要的操作行為，其實也不脫這七種。Rails 透過 HTTP 作為 generic connector interface，使用 HTTP 的四種 verb ： GET、POST、PUT、DELETE 對資源進行操作。

GET

```
<%= link_to("List", posts_path) %>
<%= link_to("Show", post_path(post)) %>
<%= link_to("New", new_post_path) %>
<%= link_to("Edit", edit_post_path(post)) %>
```

POST

```
<%= form_for @post , :url => posts_path , :html => {:method => :post} do |f| %>
```

PUT

```
<%= form_for @post , :url => post_path(@post) , :html => {:method => :put} do |f| %>
```

Destroy

```
<%= link_to("Destroy", post_path(@post), :method => :delete )
```

Form 綁定 Model Attribute 的設計

Rails 的表單欄位，是對應 Model Attribute 的：

```
<h1>New post</h1>

<%= form_for @post , :url => posts_path do |f| %>
   <%= f.error_messages %>
   <div><label>subject</label><%= f.text_field :subject %></div>
   <div><label>content</label><%= f.text_area :content %> </div>
   <%= f.submit "Submit", :disable_with => 'Submiting...' %>
<% end -%>


<%= link_to 'Back', posts_path %>
```

透過 form_for 傳送出來的表單，會被壓縮包裝成一個 parameter: params[:post],

```
def create
 @post = Post.new(params[:post])
 if @post.save
   flash[:notice] = 'Post was successfully created.'
   redirect_to post_path(@post)
 else
   render :action => "new"
 end
end
```

如此一來，原本創造新資源的一連串繁複動作就可以大幅的被簡化，開發速度達到令人驚艷的地步。

Cookies & Session

Cookies

看演唱會或是去遊樂園玩常常會發生一種情況，就是入場以後要暫時出場，這時候工作人員通常會給你蓋個手章，用來註記你曾經入場過，基本上 Cookies 的功用就是這個手章，只要使用者進到我們的網站，我們就幫他儲存一個 Cookies ，下次當使用者再度造訪時我們就可以由 Cookies 得知使用者的資訊。

有些遊樂園的手章上會標記當天的入園時間，以免有人回家不洗澡隔天又來玩一次，而 cookies 記錄這個時間的方法就是以 key/value 的形式儲存在使用者的瀏覽器中，例如：

```
{"login_time": "1990/2/1"}
```

但 Cookies 屬於沒有加密的公開檔案，所以不建議儲存敏感資料。

Session

相較於 Cookies 存在 Client 端， Session 則是存在 Server 的資料，通常與 Cookies 相呼應。

當使用者造訪我們的網站時，我們由伺服器產生 session id (32 byte long MD5 hash value)，並傳送存有這個 session id 的 cookie 給瀏覽器儲存，之後使用者造訪我們網站時，只需要比對 cookies 上的 session id 和 session 裡的 session id 就可以知道使用者身份，大部份的網站也是運用此原理實作儲存 User 登入狀態的機制。

這樣做的好處是若有人劫取到使用者的 Cookies 資料也無法得知資料內容，但是仍有 Hijacking 攻擊的疑慮，可以參考 Rails Guide 的 Security章節。

Rails session

在 Rails 中只要使用 `session[:session_name]` 的 instance method 就可以在 controller 拿取 Session 的資料，例如以下的購物車功能：

```
class ApplicationController < ActionController::Base
  def find_cart
    @cart = Cart.find_by(id: session[:cart_id])
  end
end
```

同理我們也可以將資料儲存或刪除 Session：

```
class ApplicationController < ActionController::Base
  def store_cart
    session[:cart_id] = cart.id
  end
  def reset_cart
    session.delete(:cart_id)
  end
end
```

延伸閱讀-Authenticity Token

所謂的 `authenticity_token` 就是一串隨機生成的 string ，在我們建立 Rails 的表單時， Rails 將會隨機生成一個 `authenticity_token` 存在 session 中，並且在表單的 hidden field 中也加入一樣的 `authenticity_token` ，當我們送出表單（post request）時， rails 會驗證表單的 `authenticity_token` 和 session 中的 `authenticity_token` 是否一樣，一樣才可以成功送出。

這樣做的好處是可以避免所謂的跨站請求偽造 (Cross-site Request Forgery)，又稱 CSRF。 CSRF 攻擊的原理是偽造一個 form 送 post 給伺服器，所以很有可能發生的情況是你在 A 網站有登入使用者，但是在 B 網站點了帶有 CSRF code的連結，內容是要重新設定 A 網站的使用者密碼為 12345678，此時 A 網站以為是你自己送出的請求，那麼你的密碼就被換掉了。

而 Rails 加上了 `authenticity_token` 這樣的機制就是為了阻擋 CSRF，因為 B 網站不會知道你的 `authenticity_token` 是什麼，送出的 post request 就會被擋下來。在 CSRF 中預設會檢查 POST, PUT, DELETE requests 的 authenticity_token。

參考資料：

- http://guides.rubyonrails.org/security.html#sessions
- http://guides.rubyonrails.org/action_controller_overview.html#session
- http://andikan.github.io/blog/2012/10/03/cookie-and-session/
- http://stackoverflow.com/questions/941594/understand-rails-authenticity-token

DB migration

在 Rails 的開發流程，我們會使用 `rails g model post` 產生所需要的 model 檔案。這個指令會在 `app/model/` 下產生 post.rb，這是 model 檔案。此指令另外也會在 `db/migration/` 下產生 `(time_stamp)_create_post.rb`，這是連帶產生的 db migration 檔。

在傳統的 web application 開發流程中，並不存在 db migration 機制。開發者多半使用 phpmyadmin 開設所需要的 db 欄位。

db migration 檔的格式如下：

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.string :title
      t.text :excerpt
      t.text :content
      t.timestamps
    end

  end

  def self.down
    drop_table :posts
  end
end
```

Rails 開發者將所需要開設的欄位加進 migration 檔案，執行 `rake db:migrate`，即可完成新增資料庫以及資料欄位的新增。

同樣的，若要對資料庫欄位進行更動，也是一樣透過 migration。 `rails g migration add_user_id_to_post` 會產生一個空 migration 檔，再將所需變更寫入檔案，再次執行 `rake db:migrate` 即可：

```
class AddUserIdToPost < ActiveRecord::Migration

   def change
     add_column :users, :user_id, :integer
   end

end
```

資料庫欄位變更在協同開發以及佈署上所造成的問題

由其他語言剛轉換過來的開發者，對於這樣的設計通常會感覺到不習慣。透過 phyMyAdmin 就可以進行的操作，為什麼要繞了一個圈？透過 rake 和 migration 對資料庫進行操作。

在傳統開發流程中，當一個人獨自進行開發且未進行至部署階段時，開發者都不會感覺到有什麼問題。

問題 1: 協同開發中，程式碼與 db 欄位的不一致

但若專案陸續加入了第二位、第三位開發者，就會陷入極大的麻煩：沒有人知道現在的 db schema 長成什麼樣子。當 A 決定將 post 的 title 改成 subject 時，僅透過 phyMyAdmin 改變了 db schema，而忘了告訴 B，就將 code push 到了程式版本控制系統。而 B 根本不曉得了這一個變更，於是 B 的開發端就爆炸了，這樣的事件層出不窮。

問題 2: 沒有 automation 與 rollback 機制

若沒有使用 db migration 機制，在佈署時開發者必須要自己手動輸入 SQL 或透過 phpMyAdmin 變更資料欄位。而這樣的動作在程式碼與 production 環境且 db schema 差異過大時，是非常危險的：

1. 開發者可能會因為手動操作的關係打錯字，而讓線上程式炸掉
2. 因為差異過大，網站可能會有 downtime。

3. 當變更資料庫欄位後，之後發現程式其實有問題，無法輕易的恢復剛剛所作的變更。

對 **database** 進行版本控制，流程自動化且可回復

DB schema 難道是不可能被版本控制的嗎？ Rails 解決了這樣的難題：db migration 正是因此誕生的的 best practices。

透過 db migration 檔，可以紀錄資料庫每一次的欄位變遷，並且可以被版本控制。而協同開發者，在 checkout 程式碼下來之後，看到欄位發生變更，只要執行 `rake db:migration` 就可以追上資料庫最新的進度。

同時， `rake db:migrate` 的動作是全自動化的，如果在開發端能夠執行，在 production 環境上也可以重製。而 db migration 檔也提供了回復的機制，如果程式碼哪邊出現了問題，導致對應的資料庫欄位可能也要暫時改回去，只要執行 `rake db:rollback` 就可以回復到上一次變更前的狀態。

Routing

Routing的概念就像是總機小姐，當你打電話進一間大公司時，都會有總機小姐詢問你要辦理什麼業務，然後幫你轉接給業務承辦人員，然後再由承辦人員完成你申請的業務。Routing也是做一樣的事，當你想要到某個頁面（辦理業務）時，Routing就會幫你找到負責這個業務的承辦人員（Controller Action）然後業務人員就會幫你辦完業務，給你你想要的頁面（View），大功告成。

Example:

```
#指定動作跟controller action
get 'products/:id' => 'catalog#view'

#自動按照rails內建的RESTful路徑判斷controller action
resources :products

#member & collection
resources :products do
#會產生products/:id/short
     member do
        get 'short'
        post 'toggle'
    end
#會產生products/sold
    collection do
        get 'sold'
    end
end
```

Rack

> Rack provides a minimal interface between webservers supporting Ruby and Ruby frameworks.

簡單來講 Rack 就是讓 Ruby 以及 Ruby Web 框架跟 Web Server 互動的橋樑，所有與 Server 的互動都會經過 Rack。

所以確切的說，我們幾乎在 Rails controller / route 內寫的大部分 code 都是在跟 Rack 互動，Rack會負責再幫我們跟 Server 互動，扮演秘書的角色。

對 Webserver 的開發者來說，世界上有千百種 Framework，每新增一個 Framework 就樣讓 Server 支援它會是一件很恐怖的工作，所以就需要一個共同的規範 `rack SPEC`，只要 Framework 都遵守著rack spec，server就只需要處理 Rack送來的訊號就可以了。

而這個SPEC其實就是一個帶 environment 參數的call method，由 Rack跟 server 拿這個enviroment應該要回傳的資料，並回串一個由三個主要區塊「HTTP狀態」、「HTTP Headers」、「HTTP內容」組成的字串。

而所謂的 Rack middleware 就是指在 Rack 和 Application 中間的 application，我們的每個request其實都是經過很多層的middleware才傳到Rails App中，所以有些不需要給 Rails處理的工作就可以在middleware中做，對效能有幫助。

參考資料：

- http://webcache.googleusercontent.com/search?q=cache:eBrDs9kEBksJ:wp.xdite.net/%3Fp%3D1557+&cd=1&hl=zh-TW&ct=clnk&gl=tw
- http://www.whatcodecraves.com/articles/2012/07/23/ruby-on-rack
- http://rack.github.io/
- http://guides.rubyonrails.org/rails_on_rack.html#resources
- middleware應用：http://railscasts.com/episodes/151-rack-middleware

Rake

注意喲，別把 Rake 和 Rack搞混了。

Rake 可以讓每個script之間可以有清楚的相依性，且能夠只執行程式需要執行的部分，作用大概就跟管家很像，當我們要管家去掃地時，不用告訴管家你要先去拿掃把、然後走去庭院、然後開始掃地，只要單純地說「去掃地」就好了，而在這當中「拿掃把」、「走去庭院」、「開始掃地」都是不同的script，管家（rack）會依序執行它。而且他也會記得他掃過的區域，不會每次掃地都一直重新掃一樣的地方。

例如我們可以寫一個`dev.rake`的檔案，並用`rake dev:build`的方式執行下面這段script中的build task

```
namespace :dev do
   desc "Rebuild system"
   task :build => ["tmp:clear", "log:clear", "db:drop", "db:create", "db:migrate"]
   task :rebuild => [ "dev:build", "db:seed" ]
end
```

rake就會按照順序執行完後面的一系列rake script。

Bundler

Bundler 是一套可以解決外部工具及其相依關係的好用工具，現在基本上所有以 Ruby 開發的工具，基本上都是使用這套工具管理專案上的套件相依關係。

Bundler 依據 `Gemfile` 這個檔案安裝以及判斷套件相依性。 `bundle install` 可以幫我們安裝 project 裡面需要的正確的 Gem 版本。

Rails 中的 Gemfile 內會是長這樣

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Record
gem 'sqlite3'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'

# Turbolinks makes following links in your web application faster. Read more: https://github.com/rails/turbolinks
gem 'turbolinks'

# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 1.2'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end

# Use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'

# Use unicorn as the app server
# gem 'unicorn'

# Use Capistrano for deployment
# gem 'capistrano', group: :development

# Use debugger
# gem 'debugger', group: [:development, :test]
```

常見 Bundler 指令

- `bundle check` : 檢查此專案的套件是否漏失
- `bundle install` : 安裝此專案所需要的套件

在 clone 下一個專案後，我們先會使用 `bundle install` 這個指令確定這個專案裡面，套件都被正確安裝了，才能繼續啟動專案。

I18n

I18n就是internationalization，主要的意思是讓程式可以根據使用者不同的地區顯示不同的結果，最常見於多國語系的網站中，在Rails中有一個`en.yml`就是記錄語系的檔案，我們也可以自己新增`zh-TW`的語系：

config/locales/en.yml

```
en:
  hello: "Hello world"
zh-TW:
  hello: "哈羅"
```

如此一來當使用者使用en語系時`t("hello")`就會輸出`Hello world`，切換為zh-TW語系時`t("hello")`就會輸出成`哈羅`，很方便。

Chapter3 - 關於Ruby

Ruby的作者－－松本行弘於1993年2月24日開始編寫Ruby，直至1995年12月才正式公開發佈於fj（新聞群組）。之所以稱為Ruby是取法自Perl，因為Perl的發音與6月的誕生石pearl（珍珠）相同，Ruby選擇以7月的誕生石ruby（紅寶石）命名。

Ruby相較之下比其他類似的程式語言（如Perl或Python）年輕，又因為Ruby是日本人發明的，所以早期的非日文資料和程式都比較貧乏，在網上仍然可以找到早期對Ruby的資料太少之類的批評。約於2000年，Ruby開始進入美國，英文的資料開始發展。

2004年，RoR框架誕生，Ruby更加廣為人知，Ruby並於2006年為TIOBE獲選為年度程式語言。

資料來源： http://zh.wikipedia.org/wiki/Ruby

Ruby syntax

這章將花上一些篇幅，講解初學者較不理解，但相當重要的幾個主題：

- map
- lambda
- self
- block
- symbol

map

基本上跟collect是一樣的方法，可以參考Model的collect(&:id)介紹即可。

lambda

要解釋 lambda 必須先解釋 Proc，Proc就是把block存成實例的方式，呼叫前加上 `&` 就可以轉回block：

```
p = Proc.new { |x| puts x * 2 }  # 也可以寫成 p = proc { |x| puts x * 2 }
# => #<Proc:0x007ffb8c250488@(irb):66>   存成Proc的實例
# 要用時加上&
[1,2,3].each(&p)
# => 2
# => 4
# => 6
p.class
# => Proc
```

lambda 基本上也是 Proc 的實例，只是它的形態叫做lambda。

```
lam = lambda { |x| puts x * 2 }
# => #<Proc:0x007ffb8c230408@(irb):5 (lambda)>  也是存成Proc的實例，但是存成lambda的形態
[1,2,3].each(&lam)
# => 2
# => 4
# => 6
lam.class
# => Proc
```

lambda 和 proc的主要差別有兩個

1. lambda會check參數數量，proc不會

```
# lambda
lam = lambda { |x| puts x }
lam.call(2) #=> 2
lam.call(2,3) #=> ArgumentError: wrong number of arguments (2 for 1)

# proc
proc = Proc.new { |x| puts x }
proc.call(2) #=> 2
proc.call(2,3) #=> 2   proc忽略了其他參數，只傳第一個
```

2. `return` 的處理不同，lambda的 `return` 只會跳出lambda，proc的 `return` 會跳出整個method

```
def lambda_test
  lam = lambda { return }
  lam.call
  puts "Haha! after lam call, i survived!"
end
def proc_test
  p = Proc.new { return }
  p.call
  puts "after proc call, i survived!"
end

lambda_test # => "Haha! after lam call, i survived!"
proc_test # => #沒東西，因為在執行到puts前就跳出proc_test這個method了
```

參考資料

- http://augustl.com/blog/2008/procs_blocks_and_anonymous_functions/
- http://awaxman11.github.io/blog/2013/08/05/what-is-the-difference-between-a-block/

self

self指的就是我們物件本身

舉例說明，假設我們有一個class Foo：

```
class Foo
  def self.foo
     "class method"
  end
  def foo
     "instance method"
  end
  def foobar
     self.foo
  end
end
```

我們可以用class呼叫method，self現在是Foo class：

```
Foo.foo # => "class method"
```

或是讓Foo class生成一個實例，再用實例呼叫method：

```
# 產生Foo class 的 object
f = Foo.new  # => #<Foo:0x007fd103969840>
f.foo # => "instance method"
# foobar method 裡面的 self 就是 f 這個實例
f.foobar # => "instance method"
```

此章節建議搭配instance method / class method和instance variable / class variable一起觀看。

block

Block是可以暫存一段ruby code的地方，用大括號`{}`表示 舉例說明：

```
def block_test
  puts "test start"
  yield
  puts "test done"
end

block_test{ puts "block working here!" }
# => "test start"
# => "block working here!"
# => "test done"
```

我們將程式存在block中並呼叫`block_test`方法，如此一來block的內容就會替代掉`yield`。

此外常見的`do....end`也是block的表示方法：

```
@people.each do |person|
    puts person.name
end
```

Symbol

要了解 Ruby 中的 Symbol 就必須了解物件的觀念，此篇會有簡單說明，若需要了解更詳細，坊間有許多物件導向的教學書籍可以請讀者多多參考。

何謂物件？

我們可以想像電腦擁有許多記憶體位置，就像是圖書館擁有許多書櫃，每個記憶體位置假設都可以放一個物件，就如同書櫃上的每個位置都可以放一本書。 每個物件會有特定的 id ， 就如同每本書都會有一個索引。

Ruby與物件

在 Ruby 中，所有東西都是物件，所以當我們要產生新字串的時候，其實我們就是產生了新的物件放在記憶體中，讓我們在 `irb` 中測試看看：

```
2.0.0-p481 :001 > "rails102".object_id
# => 2173297300
2.0.0-p481 :002 > "rails102".object_id
# => 2173288960
2.0.0-p481 :003 > "rails102".object_id == "rails102".object_id
# => false
```

由上方的測試中可以發現，每當我們呼叫 `"rails102"` 的字串，其實都是新增一個物件，各自擁有不同的 id ， 就像是兩本同樣名字的書，你可以說他們是同樣內容的書，但並不是「同一本」書。

Ruby Symbol 的特性

而我們再來看看 Symbol ：

```
2.0.0-p481 :001 > :rails102.object_id
# => 538408
2.0.0-p481 :002 > :rails102.object_id
# => 538408
2.0.0-p481 :003 > :rails102.object_id == :rails102.object_id
# => true
```

藉由上方的 Symbol 測試可以發現，我們每次呼叫 `:rails102` 時其實都是在叫同一個物件，所以都會有相同的 id （相對於上方字串的測試則是每次呼叫都新增一個物件）。

由於 Symbol 總是呼叫同一個物件的特性，所以很適合當作 Hash 的 key ， 也擁有較好的執行效率。

參考資料

- http://ihower.tw/rails3/ruby.html
- https://www.ruby-lang.org/zh_tw/documentation/ruby-from-other-languages/

instance method / class method

class method就是給class層級呼叫的method instance method則是給class的實例呼叫的method

舉例說明，假設我們有一個class Test長這樣：

```
class Test
# 在class內定義method時加上self代表要直接取用class
  def self.class_method
    "class method"
  end

  def instance_method
    "instance method"
  end
end
```

呼叫class method時的情況：

```
#class的呼叫
Test.class_method # => "class method"
Test.instance_method # => NoMethodError: undefined method `instance_method' for Test:Class
```

呼叫instance method時的情況：

```
#class實例的呼叫
test = Test.new  # => #<Test:0x007fd103969840>
test.class_method # => undefined method `class_method' for #<Test:0x007fd103969840>
test.instance_method # => "instance method"
```

參考資源

- http://railstips.org/blog/archives/2009/05/11/class-and-instance-methods-in-ruby

instance variable / class variable

簡單來說，class variable跟著class走，instance variable跟著實例走 class variable跟instance variable都可以被繼承 但只要是在class底下的任何實例改變了class變數，其他同個class的實例的class變數也會被改變 class variable例子：

```
class Earth
   # class variable用@@定義
  @@pollution = "low"
end

class HumanBorn < Earth
   @@pollution = "high"
end
```

讓我們查看這時Earth的變數 `@@pollution` 就變成了 `"high"`，因為是class variable所以被HumanBorn更改了。

但若class底下的實例是改變實例變數，那麼就只會在實例中被改變，不會影響到其他實例 instance variable例子：

```
class Kid
   @age = 1
  def grow_up
     @age = @age +1
  end

  def show_age
     @age
  end
end

wayne = Kid.new
wayne.show_age # => 1
wayne.grow_up
wayne.show_age # => 2

jason = Kid.new
# wayne.grow_up改變的是wayne這個實例內的instance variable，所以不影響jason這個實例
jason.show_age # => 1
```

參考資源

- http://railstips.org/blog/archives/2006/11/18/class-and-instance-variables-in-ruby/

Mixin / Extend / Inheritance

在說明Mixin跟Extend之前我們必須先解釋何為Module，因為Mixin跟Extend是在引入module時會使用到的語法，而Inheritance則是繼承class時會用到的。

Module

Module的好處就是可以選擇性的引用Module內的方法，不會讓Module內的變數或是Method與其他Class互相影響，有點類似class補充包的感覺。而引用Module的方式就是Mixin & Extend。

Mixin

引用Module的方式有以下兩種 第一是可以在Class內include Module

```
class Include < Example
  include module
end
```

使用include的方式可以讓module的method在class底下被取用 所以我們就可以使用這樣的方法：

```
user.module_method
```

Extend

另外一個引用Module方式是Extend Extend的用法是讓Class可以直接取用Module內的Method

```
class Extend < Example
  extend module
end
```

所以這樣這個時候我們就可以使用這樣的寫法：

```
Extend.module_method
```

Inheritance

Inheritance（繼承）的意思就是繼承者擁有被繼承者的特性，用於類別(class)的繼承，所以子類別可以呼叫父類別的方法

```
class Parent
  @last_name = "james"
  def laugh
    "heeeeeeeeha!"
  end
  def last_name
    @last_name
  end
end

class Child < Parent
end

dad = Parent.new
son = Child.new
#Child可以呼叫Parent的method
papa.laugh # => "heeeeeeeeha!"
son.laugh # => "heeeeeeeeha!"
#Child也繼承了Parent的實例變數
son.last_name # => "james"
```

參考資源

- http://railstips.org/blog/archives/2009/05/15/include-vs-extend-in-ruby/

override

ruby class內的method都是可以被更改的，例如我們可以更改class Array內的to_s方法：

```
# 原本的to_s method
[1, 2, 3].to_s #=> [1, 2, 3]

# 重新'打開'Array這個class
class Array
  def to_s
    "hahaha! you can't convert to string right now, uh?"
  end
end
# 再試一次to_s method
[1, 2, 3].to_s #=> "hahaha! you can't convert to string right now, uh?"
```

如此一來我們就可以靈活運用各種method了。

begin rescue

Ruby 中例外的處理方式是靠 begin-end block 的 rescue 判斷並執行。

```
begin
  # 可能會發生例外的 code
rescue AExceptionClass => some_variable
  # 屬於 AExceptionClass 的例外發生時 run 這段 code
rescue BExceptionClass => some_other_variable
  # 屬於 BExceptionClass 的例外發生時 run 這段 code
else
  # 都沒有例外發生時 run 這段 code
ensure
  # 無論有沒有發生例外，都會 run 這段 code
end
```

例外的子類別可以參考 http://www.ruby-doc.org/core-2.1.2/Exception.html

另外也可以寫成這樣：

```
def foo
  # 正常時的處理
rescue
  # 發生例外時的處理
end
```

rails中的使用

在 rails 中最常會使用到的情況是 `save!`, `create!` 這兩種指令，一般我們使用 `save` 和 `create` 只會回傳 boolean 值，但是使用 `save!` 和 `create!` 則會觸發 `ActiveRecord::RecordInvalid` 並拋出例外訊息。

ActiveRecord::RecordInvalid

```
begin
  complex_operation_that_calls_save!_internally
rescue ActiveRecord::RecordInvalid => invalid
  puts invalid.record.errors
end
```

Ruby 撰碼慣例

雖然 Ruby 是一門能夠讓開發者發揮十足創意的魔幻語言，你可以用很多種方式完成同一件事。但這個語言的圈子內的開發者，在 Coding Style 上其實還是有一定的默契存在。

本章將會介紹幾種常見慣例：

- 縮排慣例
- 命名慣例
- 迴圈慣例
- 括號慣例
- 布林慣例
- 邏輯慣例
- 字串慣例
- 陣列慣例

縮排慣例

Ruby 的縮排是 2 個空格。不是 3，也不是 4。

```
# good
def hello
  "Hello World"
end

# bad
def hello
    "Hello World"
end
```

編輯器的自動縮排功能

Sublime Text2 設定, Settings Default =>

```
// The number of spaces a tab is considered equal to
"tab_size": 2,

// Set to true to insert spaces when tab is pressed
"translate_tabs_to_spaces": true,
```

必裝 **Submlime Text2** 套件

- BeautifyRuby

`Ctrl` + `Cmd` + `k` 自動對 Ruby 縮排

命名慣例

變數或者是 method 名稱，採用 snake_case

```
def credit_card_discount
  original_price * 0.9
end
```

Class 和 Module 名稱，採用 CamelCase

```
class UserProfile
  def initialize(name)
    @name = name
  end
end
```

CONSTANT 使用 SCREAMING_SNAKE_CASE

```
class Invoice
  CREDIT_CARD_TYPE = ["VISA","MASTER"]
end
```

迴圈慣例

單行的迴圈使用 { }， 多行的迴圈使用 do end

```
10.times {|n| puts "This is #{n}"}

10.times.do |n|
  puts "This is #{n}"
  puts "That is #{n*2}"
end
```

大師 Jim Weirich 在 Braces VS DO/END 這篇提出他的觀點：

- Use { } for blocks that return values
- Use do / end for blocks that are executed for side effects

```
# block used only for side effect
list.each do |item| puts item end

# Block used to return test value
list.find { |item| item > 10 }

# Block value used to build new value
list.collect { |item| "-r" + item  }
```

使用 each 而非 for

Ruby 中跑迴圈的方式可以有很多種，最常使用的是 for 和 each

使用無窮迴圈測試四種常見迴圈寫法。

```
Benchmark.bm do |x|

  x.report('while1') do
    n = 0
    while 1 do
      break if n >= 1000000
      n += 1
    end
  end

  x.report('loop') do
    n = 0
    loop do
      break if n >= 1000000
      n += 1
    end
  end

  x.report('Infinite.each') do
    n = 0
    (0..(1/0.0)).each do
      break if n >= 1000000
      n += 1
    end
  end

  x.report('for Infinite') do
    n = 0
    for i in (0..(1/0.0)) do
      break if n >= 1000000
      n += 1
    end
  end

end
```

跑出來的數值如下

```
                 user     system      total        real
while1         31.960000   0.150000  32.110000 ( 36.749597)
loop           42.240000   0.190000  42.430000 ( 45.533708)
Infinite.each  108.400000   0.970000 109.370000 (117.421059)
for Infinite   112.070000   1.010000 113.080000 (126.712828)
```

for 明顯比 each 慢上許多。

參考資料：

- http://www.ruby-forum.com/topic/179264

括號慣例

定義 **method** 要加括號。除非沒有參數。

```
# 有括號
def find_book(author,title, options={})
  # ....
end

# 無括號
def word_count
  # ....
end
```

使用 **method** 要加括號。除非是 **statement** 或 **command**

```
# 有括號

link_to("Back", post_path(post))

# 無括號

redirect_to post_path(post)
```

statement 多數時候要加括號，但如果狀況單純可不加括號

```
# 無括號

if post.content_size > 100
  # ....
end

# 有括號

if (post.content.size > 100) && post.is_promotion?
  # ....
end
```

「使用括號」是一個相對比較好的習慣

看到這裡都開始讓人搞迷糊了。到底是什麼樣的情況要加括號？什麼樣的情況不加括號？筆者的建議是，一旦沾到相當較複雜的陳述句時，儘量加上括號。Ruby 雖然是一個相當自由的語言，可以用空格(space)或是括號(parentheses)傳入參數。但濫用空格可能造成 Ruby 在 parsing 陳述句的時候解讀異常，產出非正確的結果，這就是大家所不願意樂見的狀況了。

括號的唯一例外：**super** 與 **super()**

在 Ruby 中，唯一加括號和不加括號會有別的是 super 與 super()。它們代表了不同的意思：

super 不加括號表示呼叫父類別的同名函式，並且將本函式的所有參數傳入父類別的同名函式。

```
class Foo
  def initialize(*args)
     args.each {|arg| puts arg}
  end
end

class Bar < Foo
  def initialize( a, b, c)
    super
  end
end
```

如果執行

```
> a, b, c = *%W[a b c]
> Bar.new a, b, c
```

將會印出 `a` `b` `c`

super() 帶括號則表示呼叫父類別的同名函式，但是不傳入任何參數。

但如果是

```
class Foo
  def initialize(*args)
     args.each {|arg| puts arg}
  end
end

class Bar < Foo
  def initialize( a, b, c)
    super()
  end
end
```

執行

```
> a, b, c = *%W[a b c]
> Bar.new a, b, c
```

則不會印出任何東西。

布林慣例

布林邏輯使用 && 與 ||， 而非 and 和 or

不少人在寫邏輯判段式時，會誤以為 && 和 || 與 and / or 是等價的，因此產出不少 bug。其實 and 和 or 跟 if / else 是等價的。所以請千萬不要在 boolean 邏輯內使用 and / or。

- 被判斷為 foo = (42 && foo) / 2

```
foo = 42 && foo / 2
=> NoMethodError: undefined method '/' for nil:NilClass
```

使用 and

```
food = 42 and foo / 2
=> 21
```

and 與 or 真正的用法

and 的用法

```
next if widget = widgets.pop
```

相當於

```
widget = widgets.pop and next
```

or 的用法

```
raise "Not ready!" unless ready_to_rock?
```

相當於

```
ready_to_rock? or raise "Not ready!"
```

參考資料

- Avdi 的 Using “and” and “or” in Ruby

邏輯慣例

if / unless 的寫作觀念

Ruby 在邏輯控制的部份，除了提供 if 還提供了 unless。

unless 等於 「if not something 」等於 「if !something」。

不過雖說 Ruby 提供了 unless 這個用法，但在實務上來說，一般還是不太推薦使用 unless。除了以下幾種狀況：

當語意較適合時，使用 **unless**

```
unless content.blank?
  # ....
end
```

沒有 **else** 的時候，使用 **unless**

當沒有 else 的時候，看起來還算 OK

```
unless foo?
  # ....
end
```

但加上一個 else，看起來就不是那麼直觀了

```
unless foo?
  # ....
else
  # ....
end
```

如果專案當中有這樣的 code，相信我，換成 if 的陳述會直觀許多。

```
if !foo?
  # ....
else
  # ....
end
```

當只有一個條件時，使用 **unless** 很適合。但多個條件時，使用 **unless** 很糟糕。

```
unless foo? && baz?
  # ....
end
```

相同的，改成 if 也會直觀許多

```
if !foo? || !baz?
  # ...
end
```

if / else 的 Fail Close 安全觀念

在撰寫安全性程式碼時，使用 if 和 unless，也會產生截然不同的差別。在 Security on Rails 中有介紹 Fail open 與 Fail close 兩種觀念：

"Fail open" way, it’s bad

```
def show
  @invoice = Invoice.find(params[:id])
  unless @user.validate_code( @invoice.code )
    redirect_to :action => 'not_authorized'
  end
end
```

"Fail close" way

```
def show
  @invoice = Invoice.find(params[:id])
  if @user.validate_code( @invoice.code )
    redirect_to :action => 'authorized
  else
    redirect_to :action => 'not_authorized'
   end
end
```

使用 Fail open：「條件不成功，才不允許，不然就允許。」出包的機率遠比 Fail close：「如果條件成功，才允許進行，不然就不允許。」高的許多，這也是值得注意的地方。

三重運算子簡化 if / else

比較簡單的 if / else statement

如:

```
if some_condition
  something
else
  something_else
end
```

其實可以簡化成 :

```
some_condition ? something : something_else
```

不要濫用三重運算子

雖說 Ruby 提供這樣的簡化方式，但儘量還是不要濫用。如果超過兩層還是要將之拆開

```
# bad
some_condition ? (nested_condition ? nested_something : nested_something_else) : something_else

# good
if some_condition
  nested_condition ? nested_something : nested_something_else
else
  something_else
end
```

字串慣例

使用 `"#{}"` 而非 `+` 串接字串。

```
# bad
full_name = first_name + ' <' + last_name + '>'

# good
full_name = "#{first_name} <#{last_name}>"
```

原因：使用 string interpolation 可以讓程式顯得更直觀。而且 `+` 會產生一堆不必要的 new object。

使用 `<<` 而非 `+` 串接字串

```
html << "<h2>Post Title </h2>"
```

原因：`String#<<` 比 `String#+` 速度快的多。而且 `+` 會產生一堆不必要的 new object。

偏好使用 `" "` **(double quote)** 而非 `' '` **(single quote)** 包覆字串

```
name = "foobar"
string = "#{name}"

# => foobar

string = '#{name}'

# => #{name}
```

原因： 雖然 `' '` 和 `" "` 都可以用來宣告字串。但 `" "` 才有 string interpolation (double quote)效果。

使用 `%()` 處理需要 **string interpolation** 但同時也需要 `" "` **(double quote)** 的狀況

有時候我們不避免的需要寫出這樣的 code

```
<%= "<div class=\"name\"> #{name} </div>" %>
```

雖然可以透過將 `" "` 換成 `' '` 讓 code 不那麼粘膩。

```
<%= "<div class='name'> #{name} </div>" %>
```

但其實還有這一招使用 `%()`，可以確保事情不會變得更加複雜。

```
<%= %(<div class="name">#{name}</div>) %>
```

Array 慣例

當要宣告一個擁有多字串的 Array 陣列時，偏好使用 %w

```
array = ["A", "B", "C", "D"]

# better
array = %w(A B C D)
```

Hash 慣例

使用 `:symbool` 而非 `"string"` 作為 Hash 的 key

```
# bad
{ "foo" => "bar" }

# good
{ :foo => "bar" }
```

原因：若使用 `"string"`，Ruby 每次都會為 key 在不同的記憶體位置再產生一個新 string object。

使用 `"string"`：

```
string1 = { "foo" => "bar" }
string2 = { "foo" => "rab" }
string1.each_key {|key| puts key.object_id.to_s}
# => 2191071500
string2.each_key {|key| puts key.object_id.to_s}
# => 2191041700
```

改採用 `:symbol`：

```
string1 = { :foo => "bar" }
string2 = { :foo => "rab" }
string1.each_key {|key| puts key.object_id.to_s}
# => 304828
string2.each_key {|key| puts key.object_id.to_s}
# => 304828
```

這就是鼓勵使用 `[:symbol]` 而非`"string"` 作為 Hash 中的key的原因，在很多記憶體使用量的膨脹和浪費的 case 中，多數都是因為產生了過多不必要的 object 所造成的問題。